EE533 - Laboratory Assignment 7

Instructor: Young Cho, Ph.D.

NOTE: You must use GitHub to store all source codes and reports associated with the lab assignments. You must commit/push all modifications with detailed descriptions at least once daily while working on the lab assignment. Please submit the GitHub link and a record of your commits, along with detailed commit descriptions, to Coursistant before the due date.

1. Overview and Objectives:

In this lab, you will design and implement a custom, standalone GPU processor. You will also build tools to interpret/translate CUDA-compatible source code to execute GPU portions (NVIDIA's nvcc-generated PTX) on your custom design. The latest GPUs are designed to accelerate both graphics and AI (more specifically, Artificial Neural Networks) functions. Since the goal of this lab is to replicate designs similar to the current and emerging Cloud products, your design should be around accelerating ANN functions for solutions that integrate computer networks.

2. The Architecture:

The architecture of recent GPU modules uses the GPU's general-purpose instructions to manage memory and support AI-specific function accelerators, such as matrix-multiplication units (i.e., Tensor cores). Your GPU should have its own instruction set and its own instruction memory independent from your ARM processor. It should include instructions for data manipulation, branches, simple math operations, and specialized AI functions, such as fused multiply-accumulate for matrices using 2 or more wide registers using a tensor core-like module. Your design should support Google's BFloat16 floating-point SIMD operations and at least 64-bit wide data. You must also write scripts and other programs to verify correct operations through multiple test CUDA source codes. In this design, there is a single program counter, a single instruction stream, and one or more sets of wide registers. Each 64-bit register is treated as a packed vector of smaller data elements. A single instruction operates on all elements within the register simultaneously.

For this lab, do not worry about implementing SIMT functions; the limited hardware resources of NetFPGA will likely not allow such extensions in your final design.

The module should consist of the following components:

- **Program Counter (PC):** There should be a single program counter to run GPU program.
- **Register File:** The GPU core should address 2 or more registers (at least 64-bit wide). The registers may be shared with an AI-acceleration module for maximum matrix and/or vector operations. Each instruction specifies the data type, and the execution unit processes all 4 packed elements in parallel within a single clock cycle.
- **Execution Unit:** The GPU's execution unit may handle a minimum set of arithmetic and logical operations on the data. In modern GPUs, their execution units are often used to compute activation functions after the matrix multiplications.
- **Tensor Unit:** In addition to the execution unit, a special acceleration module for AI should be used to handle common matrix operations, including dot products. An accumulation and ReLU function may be integrated into the module to save even more time.
- **Control Unit:** The control unit manages the fetch-decode-execute cycle using a single program counter. The control unit fetches one instruction per cycle, decodes, execute, and writes back the result.
- **Thread ID control:** In CUDA, each thread has a unique identity accessed through threadIdx.x, which kernels use to index into arrays and perform per-element operations. You are free to decide how to map the threadId.x concept onto your architecture. Whichever approach you choose, clearly document how you translated the PTX into your mechanism.
- **Memory Interface (LD/ST Unit):** The GPU loads and stores 64-bit values (a full packed register) from/to a Block RAM.

3. **CUDA Kernel Requirements**
   You must write CUDA kernels using the __global__ attribute in a file called kernel.cu. These kernels will be compiled to PTX assembly using nvcc, and you will write a parser to translate the PTX output into your custom GPU opcodes. All integer kernels operate on int16_t data (4 elements packed per 64-bit register) and BFloat16 kernels operate on __nv_bfloat16 data (also 4 elements per register). We suggest support for the following operations.

   - **Vector Addition:** Compute element-wise addition of two vectors.
   - **Vector Subtraction:** Compute element-wise subtraction of two vectors.
   - **BFloat16 Vector Multiply:** Compute CUDA's built-in BFloat16 data type vector multiply.

```
#include <cuda_bf16.h>
__global__ void bf16_vector_mul(__nv_bfloat16 *a, __nv_bfloat16 *b, __nv_bfloat16 *c) {
    int idx = threadIdx.x;
    c[idx] = __hmul(a[idx], b[idx]);
}
```

   - **BFloat16 Fused Multiply-Accumulate:** BFloat16 MAC.

```
__global__ void bf16_fma(__nv_bfloat16 *a, __nv_bfloat16 *b,
                         __nv_bfloat16 *c, __nv_bfloat16 *d) {
    int idx = threadIdx.x;
    d[idx] = __hfma(a[idx], b[idx], c[idx]);
```

   - **ReLU Activation:** Compute the Rectified Linear Unit: out[i] = max(0, in[i]). You choose to have this function be part of the GPU instruction or a built in function to the tensor unit.

   You are welcome to support additional operations. Once again, keep in mind that the hardware resources on the NetFPGA are limited. Therefore, add the support for the simplest and most essential operations first.

4. **Compiler Toolchain**
   You must build a compilation pipeline that converts your CUDA kernels into your custom GPU opcodes.

   GPU Pipeline:
   Use real compilers for the heavy lifting and write Python parsers only for assembly-to-opcode translation:

   `kernel.cu` → `"nvcc -ptx -arch=sm_80"` → `kernel.ptx` (NVIDIA PTX assembly) → `your_scripts` → `gpu_program.hex` (custom GPU machine code)

   To ensure your generated opcodes work in Verilog, you can write a Python simulator to test each clock cycle.

Submission and Demonstration

- Draw a high-level design of the datapath. The figure should depict the communication between the components.
- Include the following in your report:
  - Screen Capture of Schematics, if any
  - The source codes
  - GitHub records and descriptions per team member
- A major part of the assignment will be in demonstration of your system. Please complete the process of going through your design in your YouTube demo video.